Dynamic DFS in Undirected Graphs: breaking the $O(m)$ barrier

Surender Baswana*† Shreejit Ray Chaudhury * Keerti Choudhary *‡ Shahbaz Khan *‡

Depth first search (DFS) tree is a fundamental data structure for solving various problems in graphs. It is well known that it takes $O(m+n)$ time to build a DFS tree for a given undirected graph $G=(V,E)$ on n vertices and m edges. We address the problem of maintaining a DFS tree when the graph is undergoing *updates* (insertion and deletion of vertices or edges). We present the following results for this problem.

1. *Fault tolerant DFS tree*:
   There exists a data structure of size $\tilde{O}(m)$ [1] such that given any set $\mathcal{F}$ of failed vertices or edges, a DFS tree of the graph $G \setminus \mathcal{F}$ can be reported in $\tilde{O}(n|\mathcal{F}|)$ time.

2. *Fully dynamic DFS tree*:
   There exists a fully dynamic algorithm for maintaining a DFS tree that takes worst case $\tilde{O}(\sqrt{mn})$ time per update for any arbitrary online sequence of updates.

3. *Incremental DFS tree*:
   Given any arbitrary online sequence of edge insertions, we can maintain a DFS tree in $\tilde{O}(n)$ worst case time per edge insertion.

These are the first $o(m)$ worst case time results for maintaining a DFS tree in a dynamic environment. Moreover, our fully dynamic algorithm provides, in a seamless manner, the first deterministic algorithm with $O(1)$ query time and $o(m)$ worst case update time for the dynamic subgraph connectivity, biconnectivity, and 2-edge connectivity.

*Dept. of CSE, I.I.T. Kanpur, India (www.cse.iitk.ac.in), email: {sbaswana,keerti,shahbazk}@cse.iitk.ac.in, shreejit.1@gmail.com. The preliminary version of the paper appeared in SODA 2016.

†This research was partially supported by *UGC-ISF* (the University Grants Commission of India & Israel Science Foundation) and *IMPECS* (the Indo-German Max Planck Center for Computer Science) .

‡This research was partially supported by Google India under the Google India PhD Fellowship Award.

[1] $\tilde{O}()$ hides the poly-logarithmic factors.

1 Introduction

Depth First Search (DFS) is a well known graph traversal technique. Right from the seminal work of Tarjan [39], DFS traversal has played the central role in the design of efficient algorithms for many fundamental graph problems, namely, biconnected components, strongly connected components, topological sorting [39], bipartite matching [28], dominators in directed graph [40] and planarity testing [29].

Let $G = (V, E)$ be an undirected connected graph on $n = |V|$ vertices and $m = |E|$ edges. DFS traversal of G starting from any vertex $r \in V$ produces a rooted spanning tree, called a DFS tree with r as its root. It takes $O(m + n)$ time to perform a DFS traversal and generate a DFS tree. Given any rooted spanning tree of graph G, all non-tree edges of the graph can be classified into two categories, namely, back edges and cross edges as follows. A non-tree edge is called a *back edge* if one of its endpoints is an ancestor of the other in the tree. Otherwise, it is called a *cross edge*. A necessary and sufficient condition for any rooted spanning tree to be a DFS tree is that every non-tree edge is a back edge. Thus, it can be seen that many DFS trees are possible for any given graph. However, if the traversal of the graph is performed according to the order specified by the adjacency lists of the graph, the resulting DFS tree will be unique. The ordered DFS tree problem is to compute the order in which the vertices get visited when the traversal is performed strictly according to the adjacency lists.

Most of the graph applications in real world deal with graphs that keep changing with time. These changes/updates can be in the form of insertion or deletion of vertices or edges. An algorithmic graph problem is modeled in a dynamic environment as follows. There is an online sequence of updates on the graph, and the objective is to update the solution of the problem efficiently after each update. In particular, the time taken to update the solution has to be much smaller than that of the best static algorithm for the problem. In the last two decades, many elegant dynamic algorithms have been designed for various graph problems such as connectivity [17, 26, 27, 30], reachability [35, 37], shortest path [14, 36], spanners [7, 22, 34], and min-cut [42]. Another, and more restricted, variant of a dynamic environment is the fault tolerant environment. Here the aim is to build a compact data structure, for a given problem, that is resilient to failures of vertices/edges and can efficiently report the solution for a given set of failures. There has been a lot of work in the last two decades on fault tolerant algorithms for connectivity [10, 16, 20], shortest paths [6, 12, 15], and graph spanners [8, 11].

A dynamic graph algorithm is said to be fully dynamic if it handles both insertion as well as deletion updates. A partially dynamic algorithm is said to be incremental or decremental if it handles only insertion or only deletion updates respectively. In this paper, we address the problem of maintaining a DFS tree efficiently in any dynamic environment.

1.1 Existing results on dynamic DFS

In spite of the simplicity of a DFS tree, designing any efficient parallel or dynamic algorithm for a DFS tree has turned out to be quite challenging. Reif [32] showed that the ordered DFS tree problem is a P-Complete problem. For many years, this result seemed to imply that the general DFS tree problem, that is, the computation of any DFS tree is also inherently sequential. However, Aggarwal and Anderson [2] proved that the general DFS tree problem is in *RNC* by designing a parallel randomized algorithm that takes $O(\log^3 n)$ time. Further, the fastest parallel deterministic algorithm for general DFS tree takes $O(\sqrt{n})$ time [3, 21]. Whether the general DFS tree problem is in *NC* is still a long standing open problem.

Reif [33] and later Miltersen et al. [31] proved that P-Completeness of a problem also implies hardness of the problem in the dynamic setting. The work of Miltersen et al. [31] shows that if the ordered DFS tree is updateable in $O(\mathbf{polylog}(n))$ time, then the solution of every problem in class P is updateable in $O(\mathbf{polylog}(n))$ time. In other words, maintaining the ordered DFS tree is indeed the hardest among all the problems in class P. In our view, this hardness result, which is actually for only the ordered DFS tree problem, has proved to be quite discouraging for the researchers working in the area of dynamic algorithms.

This is evident from the fact that for all the static graph problems that were solved using DFS traversal in the 1970's, none of their dynamic counterparts used a dynamic DFS tree [26, 27, 30, 35, 9, 10, 16].

Apart from the hardness of the ordered DFS tree problem in dynamic environment, very little progress has been achieved even for the problem of maintaining any DFS tree. Franciosa et al. [18] designed an incremental algorithm for a DFS tree in a DAG. For any arbitrary sequence of edge insertions, this algorithm takes $O(mn)$ total time to maintain a DFS tree from a given source. Recently Baswana and Choudhary [4] designed a decremental algorithm for a DFS tree in DAG that requires expected $O(mn \log n)$ total time. For undirected graphs, recently Baswana and Khan [5] designed an $O(n^2)$ total time incremental algorithm for maintaining a DFS tree. These algorithms are the only results known for the dynamic DFS tree problem. Moreover, none of these existing algorithms, though designed for only a partially dynamic environment, achieves a worst case bound of $o(m)$ on the update time. So the following intriguing questions remained unanswered till date:

- Does there exist any fully dynamic algorithm for maintaining a DFS tree?
- Is it possible to achieve worst case $o(m)$ update time for maintaining a DFS tree in a dynamic environment?

Not only do we answer these open questions affirmatively for undirected graphs, we also use our dynamic algorithm for DFS tree to provide efficient solutions for a couple of well studied dynamic graph problems. Moreover, our results also handle vertex updates which are generally considered harder than edge updates.

1.2 Our results

We consider a generalized notion of updates wherein an update could be either insertion/deletion of a vertex or insertion/deletion of an edge. For any set U of such updates, let $G + U$ denote the graph obtained after performing the updates U on the graph G. Our main result can be succinctly described in the following theorem.

Theorem 1.1 *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of $k \leq n$ updates, a DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.*

With this result at the core, we easily obtain the following results for dynamic DFS tree in an undirected graph.

1. *Fault Tolerant DFS tree*:

   Given any set of k failed vertices or edges and any vertex $v \in V$, we can report a DFS tree rooted at v for the resulting graph in $O(nk \log^4 n)$ time.

2. *Fully Dynamic DFS tree*:

   Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree in $O(\sqrt{mn} \log^{2.5} n)$ worst case time per update.

3. *Incremental DFS tree*:

   Given any arbitrary online sequence of edge insertions, we can maintain a DFS tree in $O(n \log^3 n)$ worst case time per edge insertion.

These are the first $o(m)$ worst case update time algorithms for maintaining a DFS tree in a dynamic environment. Recently, there has been significant work [1, 23] on establishing conditional lower bounds on the time complexity of various dynamic graph problems. A simple reduction from [1], based on the Strong Exponential Time Hypothesis (SETH), implies a conditional lower bound of $\Omega(n)$ on the update time of any fully dynamic algorithm for a DFS tree under vertex updates. We also present an unconditional lower bound of $\Omega(n)$ for maintaining a fully dynamic DFS tree explicitly under edge updates.

1.3 Applications of Fully Dynamic DFS

In the static setting, a DFS tree can be easily used to answer connectivity, 2-edge connectivity and biconnectivity queries. Our fully dynamic algorithm for DFS tree thus seamlessly solves these problems for both vertex and edge updates. Further, our result gives the first deterministic algorithm with $O(1)$ query time and $o(m)$ worst case update time for several well studied variants of these problems in the dynamic setting. These problems include dynamic subgraph connectivity [10, 16, 17, 19, 27, 30] and vertex update versions of dynamic biconnectivity [25, 24, 27] and dynamic 2-edge connectivity [27, 17, 19]. The existing results offer different trade-offs between the update time and the query time, and differ on the types (amortized or worst case) of update time and the types (deterministic or randomized) of query time. Our algorithm, in particular, improves the deterministic worst case bounds for these problems, thus emphasizing the relevance of DFS trees in solving dynamic graph problems.

1.4 Main Idea

Let T be a DFS tree of G. To compute a DFS tree of $G + U$ for a given set U of updates, the main idea is to make use of the original tree T itself. We preprocess the graph G using tree T to build a data structure $\mathcal{D}$. In order to achieve $o(m)$ update time, our algorithm makes use of $\mathcal{D}$ to create a *reduced* adjacency list for each vertex such that performing DFS traversal using these lists gives a DFS tree for $G + U$. In fact, these reduced adjacency lists are generated on the fly and are guaranteed to have only $\tilde{O}(n|U|)$ edges.

We now give an outline of the paper. In section 2, we describe the various notations used throughout the paper. Section 3 describes an algorithm to report the DFS tree after a single update in the graph. The details of the required data structure $\mathcal{D}$ are described in Section 4. Then in Section 5, we provide an overview of our algorithm for handling multiple updates, highlighting the main intuition behind our approach. Our main result (Theorem 1.1) that reports a DFS tree after any set of updates in the graph is described in Section 7. In Section 8 we convert this algorithm to fully dynamic and incremental algorithms for maintaining a DFS tree using the overlapped periodic rebuilding technique. Finally, in Section 9 and Section 10 we describe the applications and lower bounds of dynamic DFS.

2 Preliminaries

Let U be any given set of updates. We add a dummy vertex r to the given graph in the beginning and connect it to all the vertices. Our algorithm starts with any arbitrary DFS tree T rooted at r in the augmented graph and it maintains a DFS tree rooted at r at each stage. It can be observed easily that each subtree rooted at any child of r is a DFS tree of a connected component of the graph $G + U$. The following notations will be used throughout the paper.

- $T(x)$: The subtree of T rooted at vertex x.
- $path(x, y)$: Path from the vertex x to the vertex y in T.
- $dist_T(x, y)$: The number of edges on the path from x to y in T.
- $LCA(x, y)$: The lowest common ancestor of x and y in tree T.
- $N(w)$: The adjacency list of vertex w in the graph $G + U$.
- $L(w)$: The reduced adjacency list of vertex w in the graph $G + U$.
- T^* : The DFS tree rooted at r computed by our algorithm for the graph $G + U$.
- $par(w)$: Parent of w in T^*.

A subtree T' is said to be *hanging* from a path p if the root r' of T' is a child of some vertex on the path p and r' does not belong to the path p. Unless stated otherwise, every reference to a path refers to an ancestor-descendant path defined as follows:

Definition 2.1 (Ancestor-descendant path) *A path p in a DFS tree T is said to be ancestor-descendant path if its endpoints have ancestor-descendant relationship in T.*

We now state the operations supported by the data structure $\mathcal{D}$ (complete details of $\mathcal{D}$ are in Section 4). Let U below refer to a set of updates that consists of vertex and edge deletions only. For any three vertices $w, x, y \in T$, where $path(x, y)$ is an ancestor-descendant path in T the following two queries can be answered using $\mathcal{D}$ in $O(\log^3 n)$ time.

1. $Query(w, x, y)$: among all the edges from w that are incident on $path(x, y)$ in $G+U$, return an edge that is incident nearest to x on $path(x, y)$.
2. $Query(T(w), x, y)$: among all the edges from $T(w)$ that are incident on $path(x, y)$ in $G+U$, return an edge that is incident nearest to x on $path(x, y)$.

3 Handling a single update

DFS traversal has the following flexibility : when the traversal reaches a vertex, say v, the next vertex to be traversed can be *any* unvisited neighbor of v. In order to compute a DFS tree for $G + U$ efficiently, our algorithm exploits this flexibility, the original DFS tree T, and the following property of DFS traversal.

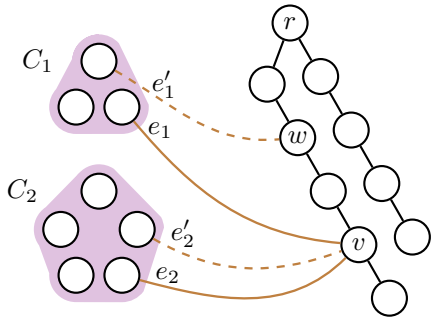

Figure 1: Edges e'_1 as well as e'_2 can be ignored during the DFS traversal.

Lemma 3.1 (Components Property) *Let T^* be the partially grown DFS tree and v be the vertex currently being visited. Let C be any connected component in the subgraph induced by the unvisited vertices. Suppose two edges e and e' from C are incident respectively on v and some ancestor (not necessarily proper) w of v in T^*. Then it is sufficient to consider only e during the rest of the DFS traversal, i.e., the edge e' need not be scanned. (Refer to Figure 1).*

Skipping e' during the DFS traversal, as stated in the components property, is justified because e' will appear as a back edge in the resulting DFS tree. In order to highlight the importance of components property, and to motivate the requirement of data structure $\mathcal{D}$, we first consider a simpler case which deals with reporting the DFS tree after a single update in the graph.

Consider the failure of a single edge (b, f) (refer to Figure 2 (i)). Exploiting the flexibility of DFS traversal, we can assume a stage in the DFS traversal of $G\backslash\{(b, f)\}$ where the partial DFS tree T^* is $T\backslash T(f)$ and the vertex b is currently being visited. Thus, the unvisited graph is a single connected component

containing the vertices of $T(f)$. Now, according to components property we need to process only the lowest edge from $T(f)$ to $path(b, r)$ ((k, b) in Figure 2 (ii)). Hence, the DFS traversal enters this component using the edge (k, b) and performs DFS traversal in the subgraph induced by the vertices of $T(f)$. The resulting DFS tree of this subgraph would now be rooted on k. Rebuilding the DFS tree after failure of the edge (b, f) thus reduces to finding the lowest edge from $T(f)$ to $path(e, r)$ and then rerooting a subtree $T(f)$ of T on the new root k. We now describe how this rerooting can be performed in $\tilde{O}(n)$ time in the following section.

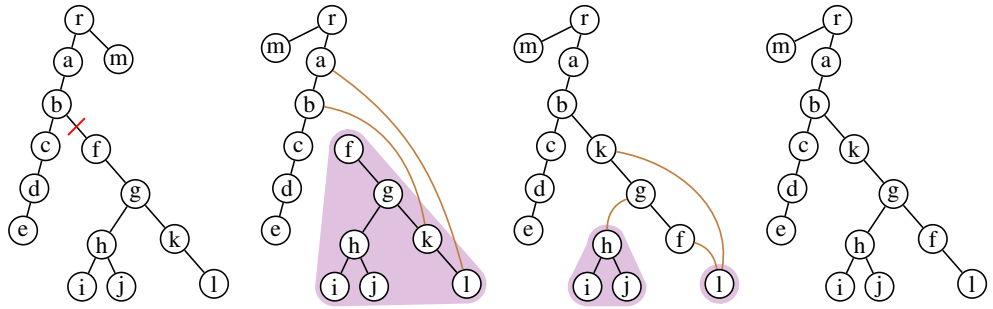

Figure 2: (i) Failure of edge (b, f). (ii) Partial DFS tree T^* with unvisited graph $T(f)$, component property allows us to neglect (a, l). (iii) Augmented $path(k, f)$ to T^*, components property allows us to neglect (l, k). (iv) Final DFS tree of $G \backslash \{(b, f)\}$.

3.1 Rerooting a DFS tree

Given a DFS tree T originally rooted at r_0 and a vertex r', the aim is to compute a DFS tree of the graph that is rooted at r'. Note that any subtree $T(x)$ of the DFS tree T is essentially the DFS tree of the subgraph induced by the vertices of $T(x)$. Hence, the same procedure can be applied to reroot a subtree $T(x)$ of the DFS tree T. Thus in general we aim at rerooting $T(r_0)$ to a new root r' (see Figure 2 (ii), where the subtree $T(f)$ would be rerooted to the new root k).

Procedure Reroot($T(r_0), r'$): Reroots the subtree $T(r_0)$ of T to be rooted at the vertex r'.

```
1  foreach (a, b) on path(r_0, r') do          /* a = par(b) in original tree T(r_0). */
2      par(a) ← b;
3      foreach child c of b not on path(r_0, r') do
4          (u, v) ← Query(T(c), r_0, b) ;      /* where u ∈ path(r_0, r') and v ∈ T(c). */
5          if (u, v) is non-null then
6              Reroot(T(c), v);
7              par(v) ← u;
8          end
9      end
10 end
```

Figure 3: The recursive algorithm to reroot a DFS tree $T(r_0)$ from the new root r'.

Our algorithm (refer to Procedure Reroot) essentially guides the DFS traversal (exploiting the flexibility of DFS) such that components of the unvisited graph can be easily identified. The components property can then be applied to each such component, processing only $O(n)$ edges to compute the rerooted DFS tree. The DFS traversal first visits the path from r' to the root of tree $T(r_0)$. This reverses $path(r_0, r')$ in the

new DFS tree T^* as now r' would be an ancestor of r_0 (see Figure 2 (iii)). Now, each subtree hanging from $path(r', r_0)$ in T forms a component of the unvisited graph. This is because, presence of any edge between these subtrees would imply a cross edge in the original DFS tree. Using the components property we know that for each of these subtrees (say T_i) we only need to process the lowest edge from T_i on the new path from r' to r_0 in T^*. Since $path(r', r_0)$ is reversed in T^*, it is equivalent to processing the highest edge e_i from T_i to the $path(r_0, r')$ in T. Recall that this query can be answered by our data structure $\mathcal{D}$ in $O(\log^3 n)$ time (refer to Section 2). Now, let v_i be the end vertex of e_i in T_i. The DFS traversal will thus visit the component (having vertices of T_i) through e_i and produce a DFS tree of the subgraph induced by T_i that is rooted on v_i. This rerooting can be performed by invoking the rerooting procedure recursively on the subtree T_i with the new root v_i.

We now analyze the total time required by Procedure Reroot to reroot a subtree T' of the DFS tree T. The total time taken by our algorithm is proportional to the number of edges processed by the algorithm. These edges include the *tree edges* that were a part of the original tree T' and the *added edges* that are returned by the data structure $\mathcal{D}$. Clearly, the number of tree edges in T' are $O(|T'|)$. Also since the added edges eventually become a part of the new DFS tree T^*, they too are bounded by the size of the tree T'. Further, the data structure $\mathcal{D}$ takes $O(\log^3 n)$ time to report each added edge. Hence the total time taken by our algorithm to rebuild T' is $O(|T'| \log^3 n)$ time. Since $\mathcal{D}$ can be built in $O(m \log n)$ time (refer to Theorem 4.1 in the Section 4), we have the following theorem.

Theorem 3.1 *An undirected graph can be preprocessed to build a data structure in $O(m \log n)$ time, such that any subtree T' of the DFS tree can be rerooted at any vertex in T' in $O(|T'| \log^3 n)$ time.*

We now formally describe how rebuilding a DFS tree after an update can be reduced to this simple rerooting procedure (see Figure 4).

1. **Deletion of an edge (u, v):**
   In case (u, v) is a back edge in T, simply delete it from the graph. Otherwise, let $u = par(v)$ in T. The algorithm finds the lowest edge (u', v') on the $path(u, r)$ from $T(v)$, where $v' \in T(v)$. The subtree $T(v)$ is then rerooted to the new root v' and hanged from u' using (u', v') in the final tree T^*.

2. **Insertion of an edge (u, v):**
   In case (u, v) is a back edge, simply insert it in the graph. Otherwise, let w be the LCA of u and v in T and v' be the child of w such that $v \in T(v')$. The subtree $T(v')$ is then rerooted to the new root v and hanged from u using (u, v) in the final tree T^*.

3. **Deletion of a vertex u:**
   Let $v_1, ..., v_c$ be the children of u in T. For each subtree $T(v_i)$, the algorithm finds the lowest edge (u'_i, v'_i) on the $path(par(u), r)$ from $T(v_i)$, where $v'_i \in T(v_i)$. Each subtree $T(v_i)$ is then rerooted to the new root v'_i and hanged from u'_i using (u'_i, v'_i) in the final tree T^*.

4. **Insertion of a vertex u:**
   Let $v_1, ..., v_c$ be the neighbors of u in the graph. Make u a child of some v_j in T^*. For each v_i, such that $v_i \notin path(v_j, r)$, let $T(v'_i)$ be the subtree hanging from $path(v_j, r)$ such that $v_i \in T(v'_i)$. Each subtree $T(v'_i)$ is then rerooted to the new root v_i and hanged from u using (u, v_i) in the final tree T^*.

In case of vertex updates, multiple subtrees may be rerooted by the algorithm. Let these subtrees be $T_1, ..., T_c$. Thus, the total time taken by our algorithm is equal to the time taken to reroot the subtrees $T_1, ..., T_c$. Using Theorem 3.1, we know that a subtree T' can be rerooted in $\tilde{O}(|T'|)$ time. Since these subtrees are disjoint, the total time taken by our algorithm to build the resulting DFS tree is $\tilde{O}(|T_1| + ... + |T_c|) = \tilde{O}(n)$. Thus we have the following theorem.

Theorem 3.2 *An undirected graph can be preprocessed to build a data structure in $O(m \log n)$ time such that after a single update in the graph, the DFS tree can be reported in $O(n \log^3 n)$ time.*

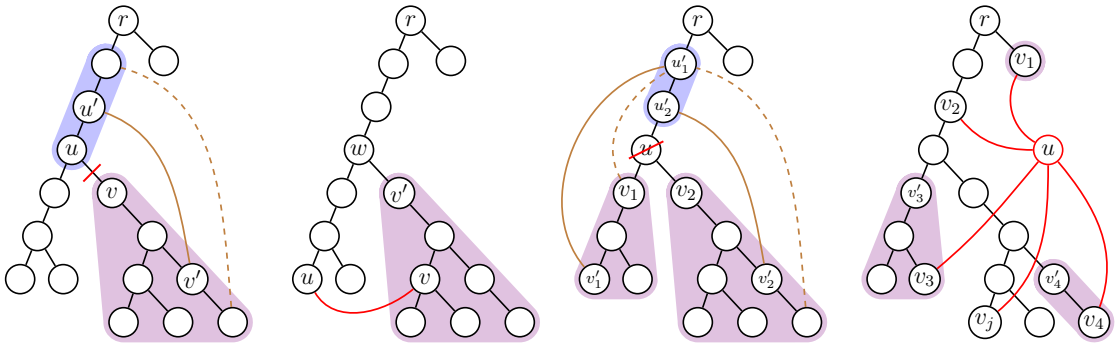

Figure 4: Updating the DFS tree after a single update: (i) deletion of an edge, (ii) insertion of an edge, (iii) deletion of a vertex, and (iv) insertion of a vertex. The algorithm reroots the marked subtrees (marked in violet) and hangs it from the inserted edge (in case of insertion) or the lowest edge (in case of deletion) on the marked path (marked in blue) from the marked subtree.

4 Data Structure

The efficiency of our algorithm relies heavily on the data structure $\mathcal{D}$. For any three vertices $w, x, y \in T$, where $path(x, y)$ is an ancestor-descendant path in T, we need to answer the following two kinds of queries.

1. $Query(w, x, y)$: among all the edges from w that are incident on $path(x, y)$ in $G+U$, return an edge that is incident nearest to x on $path(x, y)$.

2. $Query(T(w), x, y)$: among all the edges from $T(w)$ that are incident on $path(x, y)$ in $G+U$, return an edge that is incident nearest to x on $path(x, y)$.

We now describe construction of the data structure $\mathcal{D}$. It employs a combination of two well known techniques, namely, heavy-light decomposition [38] and suitable augmentation of a binary tree (segment tree) as follows.

1. Perform a pre-order traversal of the tree T with the following restriction: Upon visiting a vertex $v \in T$, the child of v that is visited first is the one storing the largest subtree. Let $\mathcal{L}$ be the list of vertices ordered by this traversal.

2. Build a segment tree $\mathcal{T}_{\mathcal{B}}$ whose leaf nodes from left to right represent the vertices in list $\mathcal{L}$.

3. Augment each node z of $\mathcal{T}_{\mathcal{B}}$ with a binary search tree $\mathcal{E}(z)$, storing all the edges $(u, v) \in E$ where u is a leaf node in the subtree rooted at z in $\mathcal{T}_{\mathcal{B}}$. These edges are sorted according to the position of the second endpoint in $\mathcal{L}$.

The construction of $\mathcal{D}$ described above ensures the following properties which are helpful in answering a query $Query(T(w), x, y)$ (see Figure 5).

- $T(w)$ is present as an interval of vertices in $\mathcal{L}$ (by step 1). Moreover, this interval can be expressed as a union of $O(\log n)$ disjoint subtrees in $\mathcal{T}_{\mathcal{B}}$ (by step 2). Let these subtrees be $T_1, \ldots, T_q$.

- It follows from the heavy-light decomposition used in step 1 that path $path(x, y)$ can be divided into $O(\log n)$ subpaths $path(x_1, y_1), \ldots, path(x_\ell, y_\ell)$ such that each subpath $path(x_i, y_i)$ is an interval in $\mathcal{L}$.

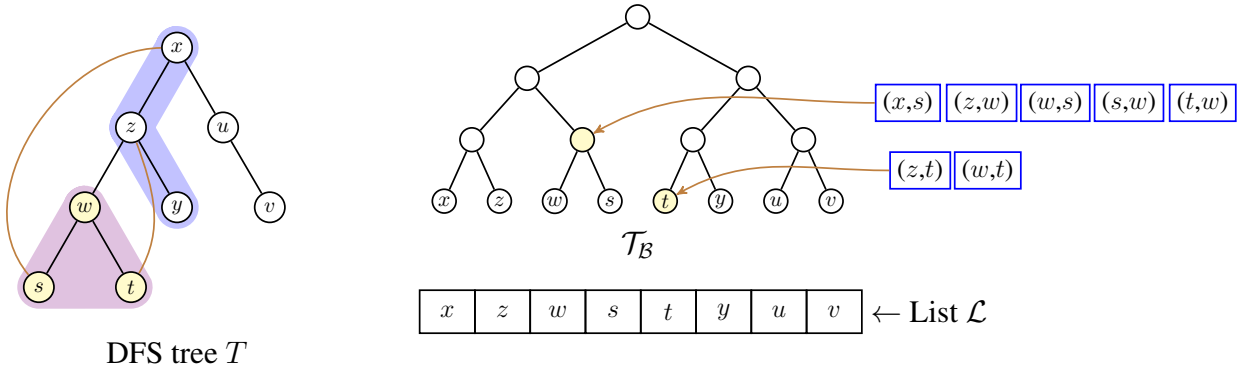

Figure 5: (i) The highest edge from subtree $T(w)$ on $path(x, y)$ is edge (x, s) and the lowest edges are edge (z, w) and (z, t). (ii) The vertices of $T(w)$ are represented as union of two subtrees in segment tree $\mathcal{T}_{\mathcal{B}}$.

- Let z_j be the root of subtree T_j in $\mathcal{T}_{\mathcal{B}}$. Then it follows from step 3 that any query $Query(T_j, x_i, y_i)$ can be answered by a single predecessor or successor query on the BST $\mathcal{E}(z_j)$ in $O(\log n)$ time.

To answer $Query(T(w), x, y)$, we thus find the edge closest to x among all the edges reported by the queries $\{Query(T_j, x_i, y_i) | 1 \leq j \leq q \text{ and } 1 \leq i \leq \ell\}$. Thus, $Query(T(w), x, y)$ can be answered in $O(\log^3 n)$ time. Notice that $Query(w, x, y)$ can be considered as a special case where $q = 1$ and T_1 is the leaf node of $\mathcal{T}_{\mathcal{B}}$ representing w. The space required by $\mathcal{D}$ is $O(m \log n)$ as each edge is stored at $O(\log n)$ levels in $\mathcal{T}_{\mathcal{B}}$. Now, the segment tree $\mathcal{T}_{\mathcal{B}}$ can be built in linear time. Further, for every node u, the sorted list of edges in $\mathcal{E}(u)$ can be computed in linear time by merging the sorted lists of its children. Thus, the binary search tree $\mathcal{E}(u)$ for each node $u \in \mathcal{T}_{\mathcal{B}}$ can be built in time linear in the number of edges in $\mathcal{E}(u)$. Hence the total time required to build this data structure is $O(m \log n)$. Thus, we have the following theorem.

Theorem 4.1 *The queries $Query(T(w), x, y)$, $Query(w, x, y)$ on T can be answered in $O(\log^3 n)$ worst case time using a data structure $\mathcal{D}$ of size $O(m \log n)$, which can be build in $O(m \log n)$ time.*

Note: Our algorithm also requires deletion of edges from $\mathcal{D}$. An edge can be deleted from $\mathcal{D}$ by deleting the edge from the binary search tree stored at its end points and their ancestors in $\mathcal{T}_{\mathcal{B}}$. Since a deletion in binary search tree takes $O(\log n)$ time, an edge can be deleted from $\mathcal{D}$ in $O(\log^2 n)$ time.

5 Handling multiple updates - Overview

DFS tree can be computed in $\tilde{O}(n)$ time after a single update in the graph, by reducing it to Procedure Reroot. However, the same procedure cannot be directly applied to handle a sequence of updates because of the following reason. The efficiency of Procedure Reroot crucially depends on the data structure $\mathcal{D}$, which is build using the DFS tree T of the original graph. Thus, when the DFS tree is updated, we are required to rebuild $\mathcal{D}$ for the updated tree. Now, rebuilding $\mathcal{D}$ is highly inefficient because it requires $O(m \log n)$ time. Thus, in order to handle a sequence of updates, our aim is to use the same $\mathcal{D}$ for handling multiple updates, without having to rebuild it after every update. We now give an overview of the algorithm that reports the DFS tree after a set U of updates.

In case of single update, all the edges reported by $\mathcal{D}$ are added to the final DFS tree T^*. However, while handling multiple updates, we use $\mathcal{D}$ to build *reduced adjacency list* for vertices of the graph, such that the DFS traversal of the graph using these *sparser* lists gives the DFS tree of the updated graph. Now, the data structure $\mathcal{D}$ finds the lowest/highest edge from a subtree of T to an ancestor-descendant path of T. Thus, in order to employ $\mathcal{D}$ to report DFS tree of $G + U$, we need to ensure that the queried subtrees and paths do

not contain any failed edges or vertices from U. Hence, for a set U of updates, we compute a partitioning of T into a disjoint collection of ancestor-descendant paths and subtrees such that none of these subtrees and paths contain any failed edge or vertex. An important property of this partitioning is that there are no edges from G lying between any two subtrees in this partitioning. We refer to this partitioning as a *disjoint tree partitioning*. Note that this partitioning depends upon only the vertex and edge failures in the set U.

Recall that during the DFS traversal we need to find the lowest edge from each component C of the unvisited graph. It turns out that any component C can be represented as a union of subtrees and ancestor-descendant paths of the original DFS tree T. The components property can now be employed to compute the reduced adjacency lists of the vertices of the graph as follows. We just find the lowest edge from each of the subtrees and the ancestor-descendant paths to T^* by querying the data structure $\mathcal{D}$. Let this edge be (x, y) where $x \in T^*$ and $y \in C$. We can just add y to the reduced adjacency list $L(x)$ of x. Since components property ensures the remaining edges to T^* can be ignored, the DFS traversal would thus consider all possible candidates for the lowest edge from every component C to T^*. Let the initial disjoint tree partitioning consists of a set of ancestor-descendant paths $\mathcal{P}$ and a set of subtrees $\mathcal{T}$. The algorithm for computing a DFS tree of $G + U$ can be summarized as follows:

Perform the static DFS traversal on the graph with the elements of $\mathcal{P} \cup \mathcal{T}$ as the super *vertices. Visiting a super vertex v^* by the algorithm involves extracting an ancestor-descendant path p_0 from v^* and attaching it to the partially grown DFS tree T^*. The remaining part of v^* is added back to $\mathcal{P} \cup \mathcal{T}$ as* new *super vertices. Thereafter, the reduced adjacency lists of the vertices on path p_0 are computed using the data structure $\mathcal{D}$. The algorithm then continues to find the next super vertex using the reduced adjacency lists and so on.*

6 Disjoint Tree Partitioning

We formally define disjoint tree partitioning as follows.

Definition 6.1 *Given a DFS tree T of an undirected graph G and a set U of failed vertices and edges, let A be a vertex set in $G + U$. The disjoint tree partitioning defined by A is a partition of the subgraph of T induced by A into*

1. *A set of paths $\mathcal{P}$ such that (i) each path in $\mathcal{P}$ is an ancestor-descendant path in T and does not contain any deleted edge or vertex, and (ii) $|\mathcal{P}| \leq |U|$.*

2. *A set of trees $\mathcal{T}$ such that each tree $\tau \in \mathcal{T}$ is a subtree of T which does not contain any deleted edge or vertex.*

Note that for any $\tau_1, \tau_2 \in \mathcal{T}$, there is no edge between τ_1 and τ_2 because T is a DFS tree.

The disjoint tree partitioning for set $A = V \backslash \{r\}$ can be computed as follows. Let V_f and E_f respectively denote the set of failed vertices and edges associated with the updates U. We initialize $\mathcal{P} = \phi$ and $\mathcal{T} = \{T(w) \mid w \text{ is a child of } r\}$. We refine the partitioning by processing each vertex $v \in V_f$ as follows (see Figure 6 (i)).

- If v is present in some $T' \in \mathcal{T}$, we add the path from $par(v)$ to the root of T' to $\mathcal{P}$. We remove T' from $\mathcal{T}$ and add all the subtrees hanging from this path to $\mathcal{T}$.

- If v is present in some path $p \in \mathcal{P}$, we split p at v into two paths. We remove p from $\mathcal{P}$ and add these two paths to $\mathcal{P}$.

Edge deletions are handled as follows. We first remove edges from E_f that don't appear in T. Processing of the remaining edges from E_f is quite similar to the processing of V_f as described above. For each edge

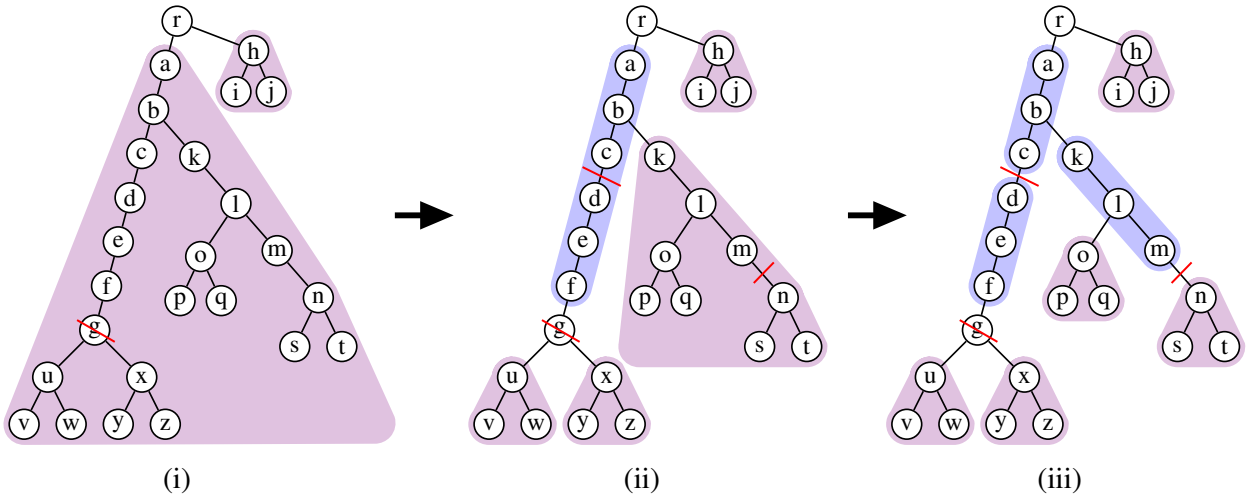

Figure 6: Disjoint tree partition for $V \setminus \{r\}$: (i) Initializing $\mathcal{T} = \{T(a), T(h)\}$ and $\mathcal{P} = \emptyset$, (ii) Disjoint tree partition obtained after deleting the vertex g. (ii) Final disjoint tree partition obtained after deleting the edges (c, d) and (m, n).

$e \in E_f$; just visualize deleting an imaginary vertex lying at mid-point of the edge e (see Figure 6 (ii)). It takes $O(n)$ time to process any $v \in V_f$ and $e \in E_f$.

Note that each update can add at most one path to $\mathcal{P}$. So the size of $\mathcal{P}$ is bounded by $|U|$. The fact that T is a DFS tree of G ensures that no two subtrees in $\mathcal{T}$ will have an edge between them. So $\mathcal{P} \cup \mathcal{T}$ satisfies all the conditions stated in Definition 6.1.

Lemma 6.1 *Given an undirected graph G with a DFS tree T and a set U of failing vertices and edges, we can find a disjoint tree partition of set $V \setminus \{r\}$ in $O(n|U|)$ time.*

7 Fault tolerant DFS Tree

We first present a fault tolerant algorithm for a DFS tree. Let U be a given set of failed vertices or edges in G. In order to compute the DFS tree T^* for $G + U$, our algorithm first constructs a disjoint tree partition $(\mathcal{T}, \mathcal{P})$ for $V \setminus \{r\}$ defined by the updates U (see Lemma 6.1). Thereafter, it can be visualized as the static DFS traversal on the graph whose (*super*) vertices are the elements of $\mathcal{P} \cup \mathcal{T}$. Note that our notion of super vertices is for the sake of understanding only.

Consider the stack-based implementation of the static algorithm for computing a DFS tree rooted at a vertex r in graph G (refer to Figure 7(i)). Our algorithm for computing DFS tree for $G + U$ (refer to Figure 7(ii)) is quite similar to the static algorithm. The only points of difference are the following.

- In the static DFS algorithm whenever a vertex is visited, it is attached to the DFS tree and pushed into the stack S. In our algorithm when a vertex u in some super vertex $v_s \in \mathcal{P} \cup \mathcal{T}$ is visited, a path starting from u is extracted from v_s and attached to the DFS tree, and this entire path is pushed into the stack S.
- Instead of scanning the entire adjacency list $N(w)$ of a vertex w, the reduced adjacency list $L(w)$ is scanned.

When a path is extracted from a super vertex v_s, the remaining unvisited part of v_s is added back to $\mathcal{T} \cup \mathcal{P}$. However, we need to ensure that the properties of disjoint tree partitioning are satisfied in the updated $\mathcal{T} \cup \mathcal{P}$. This is achieved using Procedure DFS-in-Path and Procedure DFS-in-Tree, which also build

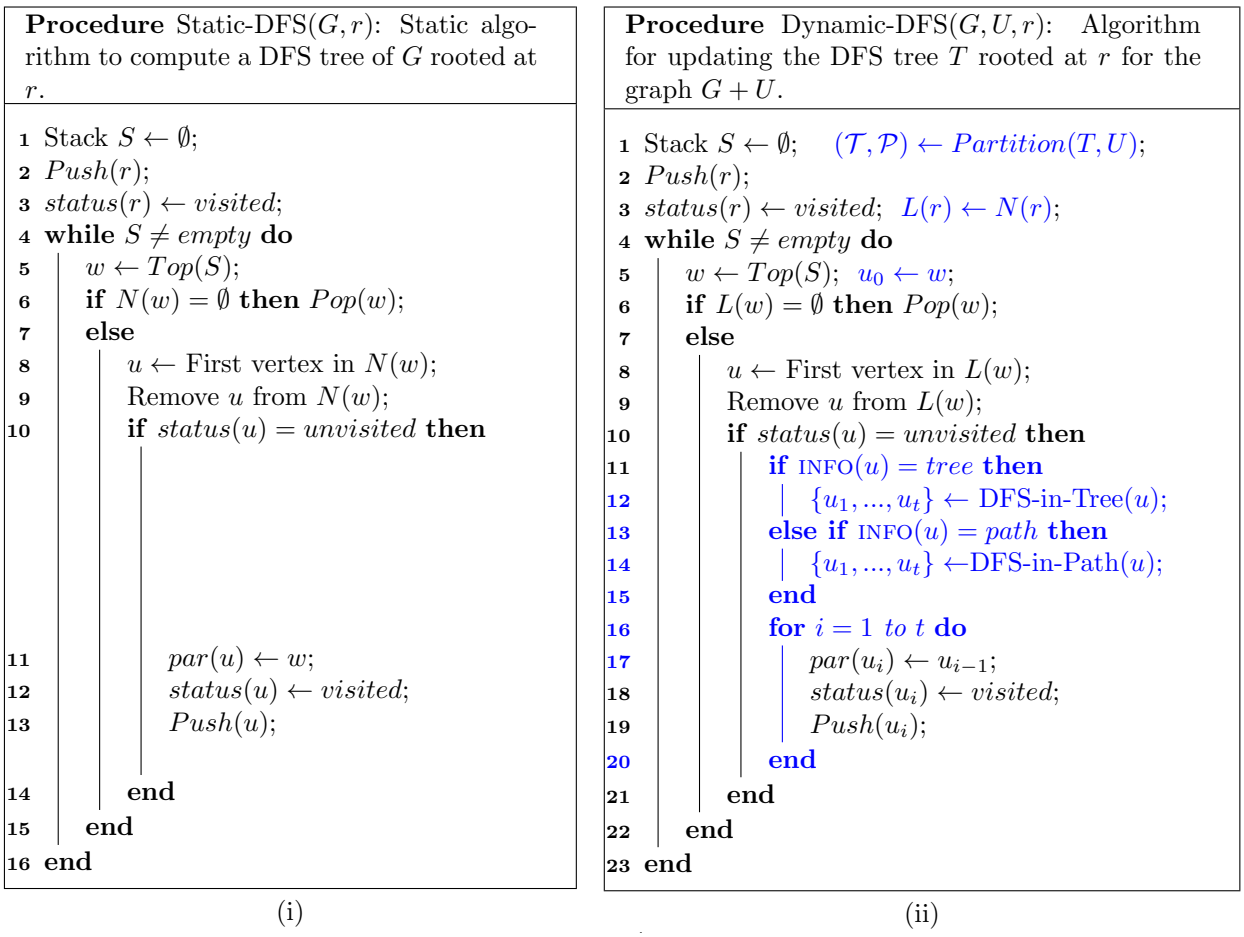

Figure 7: The static (and dynamic) algorithm for computing (updating) a DFS tree. The key differences are shown in blue.

the reduced adjacency list for the vertices on the path. The construction of a sparse reduced adjacency list is inspired by Lemma 3.1 (Component property) which can be reformulated in the context of our algorithm as follows.

Property 7.1 *When a path p is attached to the partially constructed DFS tree T^* during the algorithm, for every edge (x, y), where $x \in p$ and y belongs to the unvisited graph the following condition holds. Either y is added to $L(x)$ or y' is added to $L(x')$ for some edge (x', y') where x' is a descendant (not necessarily proper) of x in p and y' is connected to y in the unvisited graph.*

We now describe how the properties of disjoint tree partitioning and hence Property 7.1 are maintained by our algorithm when a vertex $v \in v_s$ is visited by the traversal.

1. Let $v_s = path(x, y) \in \mathcal{P}$. Exploiting the flexibility of DFS, we traverse from v to the farther end of $path(x, y)$. Now $path(x, y)$ is removed from $\mathcal{P}$ and the untraversed part of $path(x, y)$ (with length at most half of $|path(x, y)|$) is added back to $\mathcal{P}$. We refer to this as *path halving*. This technique was also used by Aggarwal and Anderson [2] in their parallel algorithm for computing DFS tree in undirected graphs. Notice that $|\mathcal{P}|$ remains unchanged or decreases by 1 after this step.

2. Let $v_s = \tau \in \mathcal{T}$. Exploiting the flexibility of a DFS traversal, we traverse the path from v to the root of τ (say x) and add it to T^*. Thereafter, τ is removed from $\mathcal{T}$ and all the subtrees hanging from this

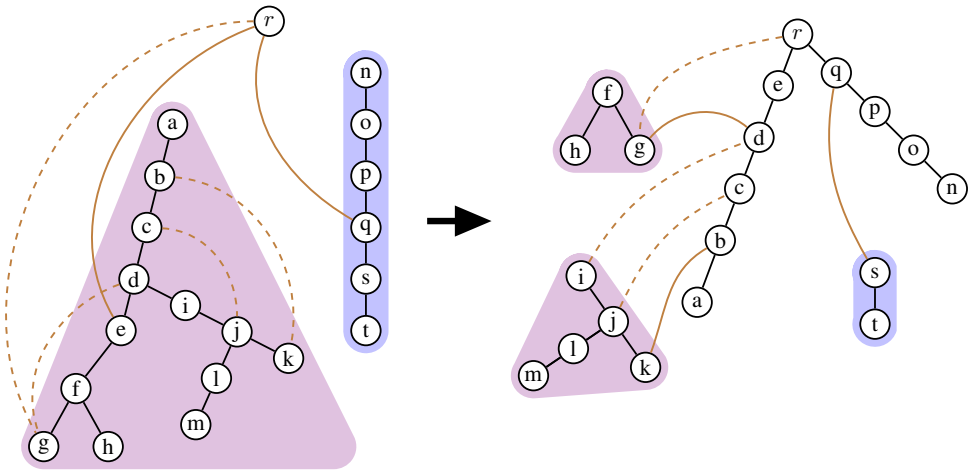

Figure 8: Visiting a super vertex from $\mathcal{T} \cup \mathcal{P}$. (i) The algorithm visits $T(a) \in \mathcal{T}$ using the edge (r, e) and the $path(n, t) \in \mathcal{P}$ using the edge (r, q). (ii) Traversal extracts $path(e, a)$ and $path(q, n)$ and augment it to T^*. The unvisited segments are added back to $\mathcal{T}$ and $\mathcal{P}$.

path are added to $\mathcal{T}$. Observe that every newly added subtree is also a subtree of the original DFS tree T. So the properties of disjoint tree partitioning are satisfied after this step as well.

Let $path(v, x)$ be the path extracted from v_s. For each vertex w in this newly added path, we compute $L(w)$ ensuring Property 7.1 as follows.

(i) For each path $p \in \mathcal{P}$, among potentially many edges incident on w from p, we just add any one edge.

(ii) For each tree $\tau' \in \mathcal{T}$, we add at most one edge to L as follows. Among all edges incident on τ' from $path(v, x)$, if (w, z) is the edge such that w is nearest to x on $path(v, x)$, then we add z to $L(w)$. However, for the case $v_s \in \mathcal{T}$, we have to consider only the newly added subtrees in $\mathcal{T}$ for this step. This is because the disjoint tree partitioning ensures the absence of edges between v_s and any other tree in $\mathcal{T}$.

Figure 8 provides an illustration of how $\mathcal{T} \cup \mathcal{P}$ is updated when a super vertex in $\mathcal{T} \cup \mathcal{P}$ is visited.

7.1 Implementation of our Algorithm

We now describe our algorithm in full detail. Firstly we delete all the failed edges in U from the data structure $\mathcal{D}$. Now, the algorithm begins with a disjoint tree partition $(\mathcal{T}, \mathcal{P})$ which evolves as the algorithm proceeds. The state of any unvisited vertex in this partition is captured by the following three variables.
-INFO(u): this variable is set to $tree$ if u belongs to a tree in $\mathcal{T}$, and set to $path$ otherwise
-ISROOT(v): this variable is set to $True$ if v is the root of a tree in $\mathcal{T}$, and $False$ otherwise.
-PATHPARAM(v): if v belongs to some path, say $path(x, y)$, in $\mathcal{P}$, then this variable stores the pair (x, y), and $null$ otherwise.

Procedure Dynamic-DFS : For each vertex v, $status(v)$ is initially set as $unvisited$, and $L(v)$ is initialized to $\emptyset$. First a disjoint tree partition is computed for the DFS tree T based on the updates U. The procedure Dynamic-DFS then inserts the root vertex r into the stack S. Now while the stack is non-empty, the procedure repeats the following steps. It reads the top vertex from the stack. Let this vertex be w. If $L(w)$ is empty then w is popped out from the stack, else let u be the first vertex in $L(w)$. If vertex u is unvisited till

Procedure DFS-in-Tree(u): DFS traversal enters from node u and exits from v, the root of the tree containing node u in set $\mathcal{T}$.

```
1  v ← u;
2  while IsRoot(v) ≠ True do
3  |  v ← par(v)
4  end
5  IsRoot(v) ← False;
6  T ← T\T(v);
7  (w_1, ..., w_t) ← path(u, v);
8  for i = 1 to t do
9  |  foreach path(x, y) ∈ P do
10 |  |  if Query(w_i, x, y) ≠ ∅ then
11 |  |  |  (w_i, z) ← Query(w_i, x, y);
12 |  |  |  L(w_i) ← L(w_i) ∪ {z};
13 |  |  end
14 |  end
15 |  foreach child w of w_i except w_{i-1} do
16 |  |  (y, z) ← Query(T(w), v, u);
   |  |  /* where y ∈ path(u, v)                */
17 |  |  L(y) ← L(y) ∪ {z};
18 |  |  T ← T ∪ T(w);
19 |  |  IsRoot(w) ← True;
20 |  end
21 end
22 Return path(u, v);
```

Procedure DFS-in-Path(u): DFS traversal enters from node u and exits from v, the farther end of path containing node u in set $\mathcal{P}$.

```
1  (v, d) ← PathParam(u);
2  if dist_T(u, d) > dist_T(u, v) then Swap(v, d);
3  c ← Neighbor of u on path(v, d) nearer to d;
4  P ← (P\path(v, d)) ∪ path(c, d);
5  for c' ∈ path(c, d) do
6  |  PathParam(c') ← (c, d);
7  end
8  (w_1, ..., w_t) ← path(u, v);
9  for i = 1 to t do
10 |  foreach path(x, y) ∈ P do
11 |  |  if Query(w_i, x, y) ≠ ∅ then
12 |  |  |  (w_i, z) ← Query(w_i, x, y);
13 |  |  |  L(w_i) ← L(w_i) ∪ {z};
14 |  |  end
15 |  end
16 end
17 foreach T(w) ∈ T do
18 |  if Query(T(w), v, u) ≠ ∅ then
19 |  |  (y, z) ← Query(T(w), v, u);
   |  |  /* where y ∈ path(u, v)                */
20 |  |  L(y) ← L(y) ∪ {z};
21 |  end
22 end
23 Return path(u, v);
```

Figure 9: The pseudo-code of Procedures DFS-in-Tree and Procedures DFS-in-Path.

now, then depending upon whether $u \in \mathcal{T}$ or $u \in \mathcal{P}$, Procedure DFS-in-Tree or DFS-in-Path is executed. A path p_0 is then returned to Procedure Dynamic-DFS where for each vertex of p_0 parent is assigned and status is marked visited. The whole of this path is then pushed into stack. The procedure proceeds to the next iteration of While loop with the updated stack.

Procedure DFS-in-Tree : Let vertex u is present in tree, say $T(v)$, in $\mathcal{T}$ (the vertex v can be found easily by scanning the ancestors of u and checking their value of IsRoot). The DFS traversal enters the tree from u and leaves from the vertex v. Let $path(u, v) = \langle w_1 = u, w_2 \ldots, w_t = v \rangle$. The $path(u, v)$ is pushed into stack and attached to the partially constructed DFS tree T^*. We now update the partition $(\mathcal{P}, \mathcal{T})$ and also update the reduced adjacency list for each w_i present on $path(u, v)$ as follows.

1. For each vertex w_i and every path $path(x, y) \in \mathcal{P}$, we perform $Query(w_i, x, y)$ on the data structure $\mathcal{D}$ that returns an edge (w_i, z) such that $z \in path(x, y)$. We add z to $L(w_i)$.

2. Recall that since subtrees in T do not have any cross edge between them, therefore, there cannot be any edge incident on $path(u, v)$ from trees which are already present in $\mathcal{T}$. An edge can be incident only from the subtrees which was hanging from $path(u, v)$. $T(v)$ is removed from $\mathcal{T}$ and all the subtrees of $T(v)$ hanging from $path(u, v)$ are inserted into $\mathcal{T}$. For each such subtree, say τ, inserted into $\mathcal{T}$, we perform $Query(\tau, u, v)$ on the data structure $\mathcal{D}$ that returns an edge, say (y, z), such that $z \in \tau$ and y is nearest to u on $path(u, v)$. We insert z into $L(y)$.

Procedure DFS-in-Path : Let vertex u visited by the DFS traversal lies on a $path(v, y) \in \mathcal{P}$. Assume $dist_T(u, v) > dist_T(u, y)$. The DFS traversal travels from u to v (the farther end of the path). The path $path(v, y)$ in set $\mathcal{P}$ is replaced by its subpath that remains unvisited. The reduced adjacency list of each $w \in path(u, v)$ is updated in similar way as in the procedure DFS-in-Tree except that in step 2, we perform $Query(\tau, u, v)$ for each $\tau \in \mathcal{T}$.

The reader may refer to Figure 9 for pseudo code of Procedures DFS-in-Tree and DFS-in-Path. This completes the description of the fault tolerant algorithm for DFS tree. This algorithm maintains Property 7.1 at each stage by construction given that the properties of disjoint tree partitioning are satisfied.

7.2 Correctness

It can be seen that the following two invariants hold for the while loop in the Procedure Static-DFS described in Figure 7 (i). It is easy to see that these invariants imply the correctness of the algorithm, i.e., the generated tree is a rooted spanning tree where every non-tree edge is a back edge.

I_1: The sequence of vertices in the stack from bottom to top constitutes an ancestor-descendant path from r in the DFS tree computed.

I_2: For each vertex v that is popped out, all vertices in the set $N(v)$ have already been visited.

These two invariants I_1 and I_2 also hold for Procedure Dynamic-DFS described in Figure 7 (ii) as follows. Invariant I_1 holds by construction as described in our algorithm. Following lemma proves that invariant I_2 is maintained by our algorithm since it follows Property 7.1 by construction.

Lemma 7.1 *If Property 7.1 is maintained by the procedure Dynamic-DFS, then invariant I_2 will hold true at each stage of the algorithm.*

Proof: We give a proof by contradiction as follows. Assume that x is the first vertex that is popped out of the stack before some vertex $y \in N(x)$ is visited. Consider the time when a path p containing x was pushed in the stack. Clearly $y \notin L(x)$, hence using Property 7.1 we know that some $y' \in L(x')$ is connected to y in the unvisited graph where x' is a descendant (not necessarily proper) of x in p. Let p^* be a path between y' and y in the unvisited graph.

Now consider the time when x is popped out of the stack. Clearly all its descendants have been popped out, so y' has been visited by the traversal. Thus, p^* can be divided into two non-empty sets A and B denoting visited and
unvisited vertices of p^* respectively. Here $y' \in A$ and $y \in B$, thus clearly for some vertex in A invariant I_2 is not satisfied. This contradicts our assumption that x is the first vertex that is popped out of the stack for which I_2 is not satisfied. Thus, maintenance of Property 7.1 ensures the invariant I_2 in our algorithm. □

Hence, our algorithm indeed computes a valid DFS tree for $G + U$.

7.3 Time complexity analysis

As described earlier the disjoint tree partitioning and the components property play a key role in the efficiency of our algorithm. They allow us to limit the size of the reduced adjacency lists L, that are built during the algorithm. Our algorithm computes T^* by performing a DFS traversal on the reduced adjacency list L. Thus, the time complexity of our algorithm is $O(n + |L|)$ excluding the time required to compute L.

We first establish a bound on the size of L. In each step our algorithm extracts a path from $v_s \in \mathcal{P} \cup \mathcal{T}$ and attaches it to T^*. Let P_t and P_p denote the set of such paths that originally belonged to some tree in $\mathcal{T}$ and some path in $\mathcal{P}$, respectively. For every path $p_0 \in P_t \cup P_p$ our algorithm performs the following queries on $\mathcal{D}$.

(i) For each vertex w in p_0, we query each path in $\mathcal{P}$ for an edge incident on the vertex w. Thus, the total number of edges added to L by these queries is $O(n|\mathcal{P}|)$.

(ii) If p_0 belongs to P_p, then we query for an edge from each $\tau \in \mathcal{T}$ to p_0. It follows from the path halving technique that each path in $\mathcal{P}$ reduces to at most half of its length whenever some path is extracted from it and attached to T^*. Hence, the size of P_p is bounded by $|\mathcal{P}| \log n$.

(iii) If p_0 belongs to P_t, then we query for an edge from only those subtrees which were hanging from p_0. Note that these subtrees will now be added to set $\mathcal{T}$. Hence, the total number of trees queried for this case will be bounded by number of trees inserted to $\mathcal{T}$. Since each subtree can be added to $\mathcal{T}$ only once, these edges are bounded by $O(n)$ throughout the algorithm.

Thus, the size of L is bounded by $O\big(n(1+|\mathcal{P}|)\log n\big)$. Since each edge added to L requires querying the data structure $\mathcal{D}$ which takes $O(\log^3 n)$ time, the total time taken to compute L is $O\big(n(1+|\mathcal{P}|\log n)\log^3 n\big)$. Thus, we have the following lemma.

Lemma 7.2 *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k failed vertices or edges (where $k \leq n$), the DFS tree of $G + U$ can be reported in $O(n(1 + |\mathcal{P}| \log n) \log^3 n)$ time.*

From Definition 6.1 we have that $|\mathcal{P}|$ is bounded by $|U|$. Thus we have the following theorem.

Theorem 7.1 *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k failed vertices or edges (where $k \leq n$), the DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.*

It can be observed that Theorem 7.1 directly implies a data structure for fault tolerant DFS tree.

7.4 Extending the algorithm to handle insertions

In order to update the DFS tree, our focus has been to restrict the number of edges that are processed. For the case when the updates are deletions only, we have been able to restrict this number to $O(nk \log n)$, for a given set of k updates (failure of vertices or edges). We now describe the procedure to handle vertex and edge insertions. Let V_I be the set of the vertices inserted, and E_I be the set of edges inserted. (including the edges incident to the vertices in V_I). If there are k vertex insertions, the size of E_I is bounded by nk. So even if we add all the edges in E_I to the reduced adjacency lists, the size of L would still be bounded by $O(nk \log n)$. Hence, we perform the following two additional steps before starting the DFS traversal.

- Initialize $L(v)$ to store the edges in E_I instead of $\emptyset$. That is, $L(v) \leftarrow \{y \mid (y, v) \in E_I\}$
- Each newly inserted vertex is treated as a singleton path and added to $\mathcal{P}$. That is, $\mathcal{P} \leftarrow \mathcal{P} \cup \{x | x \in V_I\}$.

In order to establish that our algorithm, after incorporating the insertions, correctly computes a DFS tree of $G + U$, we need to ensure that all the edges *essential* for DFS traversal as described in Property 7.1 are added to L. All the essential edges from G are added to L during the algorithm itself. In case an essential edge belongs to E_I, the edge has already been added to L during its initialization. Note that the time taken by our algorithm remains unchanged since the size of L remains bounded by $O(nk \log n)$. This completes the proof of our main result stated in Theorem 1.1.

Let us consider the case when U consists of edge insertions only. In this case $\mathcal{P}$ will be an empty set. As discussed above, we initialize the reduced adjacency lists using E_I whose size is equal to $|U|$. Hence, Lemma 7.2 implies the following theorem.

Theorem 7.2 *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k edge insertions (where $k \leq n$), the DFS tree of $G + U$ can be reported in $O(n \log^3 n)$ time.*

8 Fully dynamic DFS

We now describe the overlapped periodic rebuilding technique to convert our algorithm for computing a DFS tree after k updates to fully dynamic and incremental algorithms for maintaining a DFS tree. Similar technique was used by Thorup [41] for maintaining fully dynamic all pairs shortest paths.

In the fully dynamic model, we need to report the DFS tree after every update in the graph. Given the data structure $\mathcal{D}$ built using the DFS tree of the graph G, we are able to report the DFS tree of $G+U$ after $|U| = k$ updates in $\tilde{O}(nk)$ time. This becomes inefficient if k becomes large. Rebuilding $\mathcal{D}$ after every update is also inefficient as it takes $\tilde{O}(m)$ time to build $\mathcal{D}$. Thus, it is better to rebuild $\mathcal{D}$ after every $|U'| = c$ updates for a carefully chosen c. Let $\mathcal{D}'$ be the data structure built using the DFS tree of the updated graph $G+U'$ with $|U'| = c$. $\mathcal{D}'$ can thus be used to process the next c updates efficiently (see Figure 10 (a)). The cost of building $\mathcal{D}'$ can thus be amortized over these c updates.

To achieve an efficient worst case update time, we divide the building of $\mathcal{D}'$ over the first c updates. This $\mathcal{D}'$ is then used by our algorithm in the next c updates, during which a new $\mathcal{D}''$ is built in a similar manner and so on (see Figure 10 (b)). The following lemma describes how this technique can be used in general for any dynamic graph problem. For notational convenience we denote any function $f(m,n)$ as f.

Lemma 8.1 *Let D be a data structure that can be used to report the solution of a graph problem after a set of U updates on an input graph G. If D can be build in $O(f)$ time and the solution for graph $G+U$ can be reported in $O(h+|U|\times g)$ time, then D can be used to report the solution after every update in worst case $O(\sqrt{fg}+h)$ update time, given that $\sqrt{f/g} \leq n$.*

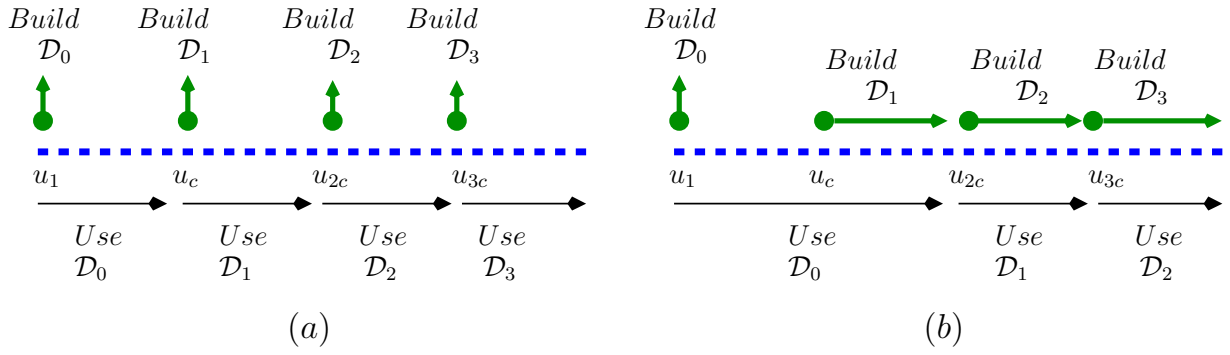

Figure 10: (a) Fully dynamic algorithm with amortized update time. (b) De-amortization of the algorithm.

Proof: We first present an algorithm that achieves amortized $O(\sqrt{fg}+h)$ update time. It is based on the simple idea of periodic rebuilding. Given the input graph G_0 we preprocess it to compute the data structure D_0 over it. Now let $u_1, ..., u_c$ ($c \leq n$) be the sequence of first c updates on G_0. To report the solution after i^{th} update we use D_0 to compute the solution for $G_0 + \{u_1, ..., u_i\}$. This takes $O(h + (i \times g))$ time. So the total time for preprocessing and handling the first c updates is $O(f + \sum_{i=1}^{c} h + (i \times g))$. Therefore, the average time for the first c updates is $O(f/c + c \times g + h)$. Minimizing this quantity over c gives the optimal value $c_0 = \sqrt{f/g}$ which is bounded by n. So, after every c_0 updates we rebuild our data structure and use it for the next c_0 updates (see Figure 10(a)). Substituting the value of c_0 gives the amortized time complexity as $O(\sqrt{fg} + h)$.

The above algorithm can be de-amortized as follows. Let $G_1, G_2, G_3, \ldots$ be the sequence of graphs obtained after $c_0, 2c_0, 3c_0, ..$ updates. We use the data structure D_0 built during preprocessing to handle the first $2c_0$ updates. Also after the first c_0 updates we start building the data structure D_1 over G_1. This D_1 is built in c_0 steps, thus the extra time spent per update is $f/c_0 = O(\sqrt{fg})$ only. We use D_1 to handle the next c_0 updates on graph G_2, and also in parallel compute the data structure D_2 over the graph G_2. (See Figure

10(b)). Since the time for building each data structure is now divided in c_0 steps, we have that the worst case update time as $O(\sqrt{fg} + h)$. □

The above lemma combined with Theorems 1.1 and 7.2 directly implies the following results for the fully dynamic DFS tree problem and the incremental DFS tree problem, respectively.

(For the theorem below we use $f = m \log n$, $g = n \log^4 n$ and $h = 0$.)

Theorem 8.1 *There exists a fully dynamic algorithm for maintaining a DFS tree in an undirected graph that uses $O(m \log n)$ preprocessing time and can report a DFS tree after each update in the worst case $O(\sqrt{mn} \log^{2.5} n)$ time. An update in the graph can be insertion / deletion of an edge as well as a vertex.*

(For the theorem below we use $f = m \log n$, $g = \log n$ and $h = n \log^3 n$, since $O(n \log^3 n) = O(k \times g + h)$ for $k \leq n$).

Theorem 8.2 *There exists an incremental algorithm for maintaining a DFS tree in an undirected graph that uses $O(m \log n)$ preprocessing time and can report a DFS tree after each edge insertion in the worst case $O(n \log^3 n)$ time.*

9 Applications

Our fully dynamic algorithm for maintaining a DFS tree can be used to solve various dynamic graph problems such as dynamic subgraph connectivity, biconnectivity and 2-edge connectivity. Note that these problems are solved trivially using a DFS tree in the static setting. Let us now describe the importance of our result in the light of existing results for these problems.

9.1 Existing Results

The dynamic subgraph connectivity problem is defined as follows. Given an undirected graph, the status of any vertex can be switched between *active* and *inactive* in an update. The goal is to efficiently answer any online connectivity query on the subgraph induced by the active vertices. This problem can be solved by using dynamic connectivity data structures [17, 19, 27, 30] that answer connectivity queries under an online sequence of edge updates. This is because switching the state of a vertex is equivalent to $O(n)$ edge updates. Chan [9] introduced this problem and showed that it can be solved more efficiently. He gave an algorithm using FMM (fast matrix multiplication) that achieves $O(m^{0.94})$ amortized update time and $\tilde{O}(m^{1/3})$ query time. Later Chan et al. [10] presented a new algorithm that improves the amortized update time to $\tilde{O}(m^{2/3})$. They also mentioned the following among the open problems.

1. Is it possible to achieve constant query time with worst case sublinear update time ?
2. Can non trivial updates be obtained for richer queries such as counting the number of connected components ?

Duan [16] answered the first question affirmatively but at the expense of a much higher update time. He presented an algorithm with $O(m^{4/5})$ worst case update time and $O(m^{1/5})$ update time, improving the worst case bounds for the problem. Kapron et al. [30] presented a randomized algorithm for fully dynamic connectivity which takes $\tilde{O}(1)$ time per update and answers the query correctly with high probability in $\tilde{O}(1)$ time, giving a Monte Carlo algorithm for subgraph connectivity with worst case $\tilde{O}(n)$ update time. Thus their result answered the first question in a randomized setting. However, in the deterministic setting both these questions were still open. Our result answers both these questions affirmatively for the deterministic setting as well. Our fully dynamic algorithm directly provides an $\tilde{O}(\sqrt{mn})$ update time and $O(1)$

query time algorithm for the dynamic subgraph connectivity problem. Our algorithm maintains the number of connected components simply as a byproduct. In fact, our fully dynamic algorithm for DFS tree solves a generalization of dynamic subgraph connectivity - in addition to just switching the status of vertices, it allows insertion of new vertices as well. Hence the existing results offer different trade-offs between the update time and the query time, and differ on the types (amortized or worst case) of update time and the types (deterministic or randomized) of query time. Our algorithm, in particular, improves the deterministic worst case bounds for the problem (see Figure 11). Further, unlike all the previous algorithms for dynamic subgraph connectivity, which use heavy machinery of existing dynamic algorithms, our algorithm is arguably much simpler and self contained.

References	**Update Time**	**Query Time**
Frederickson [19] (1985), Eppstein et. al [17] (1997) †	$O(n\sqrt{n})$	$O(1)$
Holm et al. [27] (2001) $*\dagger$	$\tilde{O}(n)$ amortized	$\tilde{O}(1)$
Chan [9] (2006)	$\tilde{O}(m^{0.94})$ amortized	$\tilde{O}(m^{1/3})$
Chan et al. [10] (2008)	$\tilde{O}(m^{2/3})$ amortized	$\tilde{O}(m^{1/3})$
Duan [16] (2010)	$\tilde{O}(m^{4/5})$	$\tilde{O}(m^{1/5})$
Kapron et al. [30] (2013)	$\tilde{O}(n)$	$\tilde{O}(1)$ (Monte Carlo)
New	$\tilde{O}(\sqrt{mn})$	$O(1)$

Figure 11: Current-state-of-the-art of the algorithms for the dynamic subgraph connectivity.

Exploiting the rich structure of DFS trees, we also obtain $\tilde{O}(\sqrt{mn})$ update time algorithms for dynamic biconnectivity and dynamic 2-edge connectivity under vertex updates in a seamless manner. These problems have mainly been studied in the dynamic setting under edges updates only. Some of these results also allow insertion and deletion of isolated vertices. Our result, on the other hand does not impose any such restriction on insertion or deletion of vertices. Figure 12 illustrates our results and the existing results in the right perspective.

References	**Update Time**	**Query Time**
Frederickson [19] (1985), Eppstein et. al [17] (1997) †	$O(n\sqrt{n})$	$O(1)$
Henzinger [25] (2000) $*$	$\tilde{O}(n\sqrt{n})$	$O(1)$
Holm et al. [27] (2001) $*\dagger$	$\tilde{O}(n)$ amortized	$\tilde{O}(1)$
New $*\dagger$	$\tilde{O}(\sqrt{mn})$	$O(1)$

Figure 12: Current-state-of-the-art of the algorithms for the dynamic biconnectivity $(*)$ and dynamic 2-edge connectivity $(\dagger)$ under vertex updates.

We now describe how our algorithm can be used to solve these problems.

9.2 Algorithm

The solution of dynamic subgraph connectivity follows seamlessly from our fully dynamic algorithm as follows. As mentioned in Section 2, we maintain a DFS tree rooted at a dummy vertex r, such that the subtrees hanging from its children corresponds to connected components of the graph. Hence, the connectivity query for any two vertices can be answered by comparing their ancestors at depth two (i.e. children of r). This

information can be stored for each vertex and updated whenever the DFS tree is updated. Thus, we have a data structure for subgraph connectivity with worst case $\tilde{O}(\sqrt{mn})$ update time and $O(1)$ query time. Our algorithm fully dynamic DFS can be extended to solve fully dynamic biconnectivity and 2-edge connectivity under both edge updates and vertex updates as follows.

A set S of vertices in a graph is called a *biconnected component* if it is maximal set of vertices such that on failure of any vertex w in S, the vertices of $S \setminus \{w\}$ remains connected. Similarly, a set S is said to be *2-edge connected component* if it is maximal set of vertices such that failure of any edge with both end points in S does not disconnect any two vertices in S. The goal is to maintain a data structure that can efficiently answer the queries of biconnectivity and 2-edge connectivity in the dynamic setting.

These queries can be answered easily by finding *articulation points* and *bridges* of the graph. It can be shown [13] that two vertices belong to same biconnected component if and only if the path connecting them in a DFS tree of the graph does not passes through an *articulation point*. Similarly, two vertices belong to same 2-edge connected component if and only if the path connecting them in a DFS tree of the graph does not have a *bridge*. The articulation point and bridge in a graph can be defined as follows:

Definition 9.1 *Given a graph $G(V, E)$, a vertex $v \in V$ is called an articulation point of G if there exist a pair of vertices $x, y \in V$ such that every path between x and y in G passes through v.*

Definition 9.2 *Given a graph $G(V, E)$, an edge $e \in E$ is called a bridge of G if there exist a pair of vertices $x, y \in V$ such that every path between x and y in G passes through e.*

The articulation points and bridges of a graph can be easily computed by using DFS traversal of the graph. Given a DFS tree T of an undirected graph G, we can index the vertices in order they were visited by the DFS traversal. This index is called the *DFN number* of the vertex. The *high number* of a vertex v is defined as the lowest DFN number vertex from which there is an edge incident to $T(v)$. Now any non-root vertex v will be an articulation point of the graph if high number of at least one of its children is equal to $DFN(v)$. The root r of the DFS tree T will be an articulation point if it has more than one child. Any edge (x, y) (say $x = par(y)$) of the DFS tree will be a bridge if the high number of y is $DFN(x)$ and the high number of each child of y (if any) is equal to $DFN(y)$. Thus, given the high number of each vertex in the DFS tree, the articulation points and bridges can be determined in $O(n)$ time.

We now show that given any set of k updates to graph G, we can not only construct the new tree T^*, but also compute high point of each vertex in $O(nk \log^4 n)$ time. For each vertex x, let $a(x)$ denote the highest ancestor of x in T^* such that $(x, a(x))$ is an edge in $G + U$. Note that if $(x, a(x))$ is a newly added edge, then it can be easily computed by scanning all the new edges added to the graph. This is due to fact that the total number of new edges added to G is bounded by nk. So we restrict ourselves to the case when $(x, a(x))$ was originally present in the graph G. Recall that our algorithm computed T^* by attaching paths to the partially grown tree, where P_t and P_p are the set of paths attached to T^* (during its construction) that originally belonged to $\mathcal{T}$ and $\mathcal{P}$ respectively. Also, path halving ensures that the size of P_p is bounded by $k \log n$. For each path $p_0 \in P_t \cup P_p$, let $H(p_0)$ denote the vertex in p_0 that is closest to r in T^*.

We now give an $O(nk \log^4 n)$ time algorithm for constructing a subset $A(x)$ of neighbors of x such that the following condition holds.

- For a vertex x, if $a(x) \notin A(x)$, then there is some descendant y of x in T^* such that $a(x) \in A(y)$.

It is easy to see that if we get such an $A(x)$ corresponding to each x, then high number of each vertex can be computed easily by processing the vertices of T^* in bottom-up manner. Now depending upon whether paths containing x and $a(x)$ belong to set P_p or P_t, we can have different cases. We show the construction of set $A(x)$ along with the case it handles in following four steps.

1. Vertex $a(x)$ lies on a path in P_p

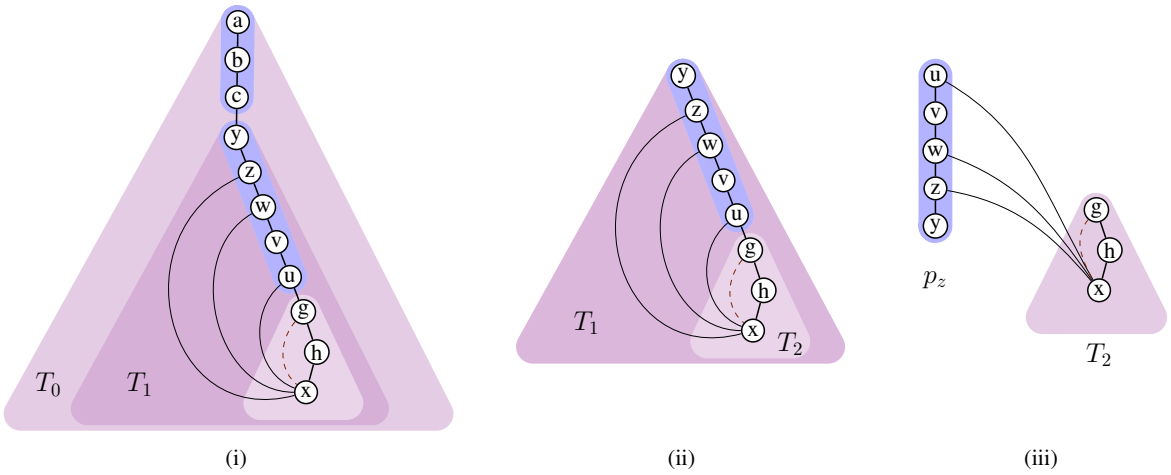

Figure 13: (i) Before the beginning of algorithm vertex x belongs to tree $T_0 \in \mathcal{T}$, z is the highest ancestor of x in T_0 such that (x, z) is an edge. (ii) The partitioning changes as the algorithm proceeds, $T_1 (\in \mathcal{T})$ is the tree containing vertex z just before it is attached to T^*. (iii) A path containing vertex z (i.e. p_z) is extracted from T_1 and attached to T^*. If $a(x)$ belongs to T_0, then it is the highest neighbor of x in p_z.

Consider a vertex x in T^*. For each path $p_0 \in P_p$ we query our data structure for an edge (u, x) such that the end point u is closest to $H(p_0)$ on path p_0, and add u to $A(x)$. Note that if $a(x)$ lies on p_0, then u will be same as $a(x)$.

2. Vertex x lies on a path in P_p

   For each $u \in T^*$ and $p_0 \in P_p$, we query our data structure for an edge (u, y) such that the end point y is farthest from $H(p_0)$ on path p_0. We add u to $A(y)$. Now consider a vertex x on p_0 such that $a(x) = u$. If x is equal to y, then we have added $a(x)$ (i.e. u) to $A(x)$. If x is not equal to y, then we have added $a(x)$ (i.e. u) to $A(y)$ where y is descendant of x in T^*.

3. Vertex x and $a(x)$ lies on same path in P_t

   Consider a vertex x lying on path $p_0 \in P_t$. We query our data structure for an edge (u, x) such that the end point u is closest to $H(p_0)$ on path p_0, and add u to $A(x)$. Note that if $a(x)$ also lies on p_0, then u will be same as $a(x)$.

4. Vertex x and $a(x)$ lies on different paths in P_t

   We first note that x and $a(x)$ would belong to same tree (say T_0) in $\mathcal{T}$, since disjoint tree partitioning ensures the absence of edges between two subtrees in $\mathcal{T}$. Let z be the highest ancestor of x in T_0 such that (x, z) is an edge in $G + U$. Let p_z be the path in P_t containing vertex z.

   We now prove that $a(x)$ belongs to p_z. Recall that as the algorithm proceeds, our partitioning $\mathcal{P} \cup \mathcal{T}$ evolves with time. Let T_1 be the tree in $\mathcal{T}$ containing vertex z just before p_z is attached to T^*. Then T_1 is either same as T_0, or a subtree of T_0 (see Figure 13 (i)). Also, $a(x)$ must lie in tree T_1, since it cannot be an ancestor of z in T_0. Now let T_2 be the tree containing x which is obtained on removal of p_z from T_1. Since z is an ancestor of x in T_0, the vertices in T_2 will eventually hang from some descendant of z (not necessarily proper) in T^*. For $a(x)$ to be the highest neighbor of x in T^*, it should be an ancestor of z in T^* which is only possible if $a(x) \in p_z$.

   Therefore, for each vertex x belonging to a tree T_0 in $\mathcal{T}$, we calculate the highest ancestor of z of x in T_0 which has an edge incident from T_0. Now once T^* is constructed let $p_z (\in P_t)$ be the path containing vertex z. We query our data structure for an edge (u, x) such that the end point u is closest

to $H(p_z)$ on path p_z, and add u to $A(x)$. Note that if $a(x)$ also lies in T_0, then u must be same as $a(x)$ (see Figure 13 (iii)).

Note that the total time taken by the first two steps is bounded by $O(nk \log^4 n)$ and by the last two steps is bounded by $O(n \log^3 n)$. Thus, we have the following theorem.

Theorem 9.1 *Given an undirected graph $G(V, E)$ with $|V| = n$ and $|E| = m$, we can maintain a data structure for answering queries of biconnected components and 2 edge connectivity in a dynamic graph which takes $O(\sqrt{mn} \log^{2.5} n)$ update time, $O(1)$ query time and $O(m \log n)$ time for preprocessing.*

10 Lower Bounds

We now prove two conditional lower bounds for maintaining DFS tree under vertex updates and edge updates.

10.1 Vertex Updates

The lower bound for maintaining DFS tree under vertex updates is based on Strong Exponential Time Hypothesis (SETH) as defined below:

Definition 10.1 (SETH) *For every $\epsilon > 0$, there exists a positive integer k, such that SAT on $k-$CNF formulas on n variables cannot be solved in $\tilde{O}(2^{(1-\epsilon)n})$ time.*

Given an undirected graph G on n vertices and m edges in a dynamic environment (incremental / decremental or fully dynamic) under vertex updates. The status of any vertex can be switched between *active* and *inactive* in an update. The goal of subgraph connectedness is to efficiently answer whether the subgraph induced by active vertices is connected. Abboud and Williams[1] proved a conditional lower bound of $\Omega(n)$ per update based on SETH for answering dynamic subgraph connectedness queries. They proved that any algorithm for answering dynamic subgraph connectedness queries using arbitrary polynomial preprocessing time and $O(n^{1-\epsilon})$ amortized update time would essentially refute the SETH conjecture. They also proved that any algorithm for maintaining partially dynamic (incremental/decremental) subgraph connectedness using arbitrary polynomial preprocessing time and $O(n^{1-\epsilon})$ worst case update time would essentially refute the SETH conjecture.

We present a reduction from subgraph connectedness to maintaining DFS tree under vertex updates requiring the algorithm to report whether the number of children of the root in any DFS tree of the subgraph is greater than 1. Thus we establish the following:

Theorem 10.1 *Given an undirected graph G with n vertices and m edges undergoing vertex updates, an algorithm for maintaining DFS tree (that can report the number of children of the root in the DFS tree) with preprocessing time $p(m, n)$, update time $u(m, n)$ and query time $q(m, n)$ would imply an algorithm for subgraph connectedness with preprocessing time $p(m + n, n)$, update time $u(m + n, n)$ and query time $q(m + n, n)$.*

Proof: Given the graph G for which we need to query for subgraph connectedness, we make a graph G' as follows. We add all vertices and edges of G to G'. Further add another vertex r called as *pseudo root* and connect it to all other vertices of G'. Thus G' has $n + 1$ vertices and $m + n$ edges. Now in any DFS tree T of G' rooted on r, the number of children of r will be equal to the number of components in G. Here subtrees rooted on each child of s represents a component of G. Any change on G can be performed on G' and query for subgraph connectedness in G is equivalent to querying if r has more than 1 child in T. □

Thus any algorithm for maintaining fully dynamic DFS under vertex updates with arbitrary preprocessing time and $O(n^{1-\epsilon})$ amortized update time would refute SETH. Also any algorithm for maintaining partially dynamic DFS under vertex updates with arbitrary preprocessing time and $O(n^{1-\epsilon})$ worst case update time would refute SETH.

10.2 Edge Updates

We now present a lower bound for maintaining a DFS tree under edge updates that applies on any algorithm which maintains tree edges of the DFS tree explicitly. In the following example we prove that there exists a graph G and a sequence of edge updates U, such that any DFS tree of the graph would require a conversion of $\Omega(n)$ edges from tree edges to back edges and vice-versa after every pair of updates in U.

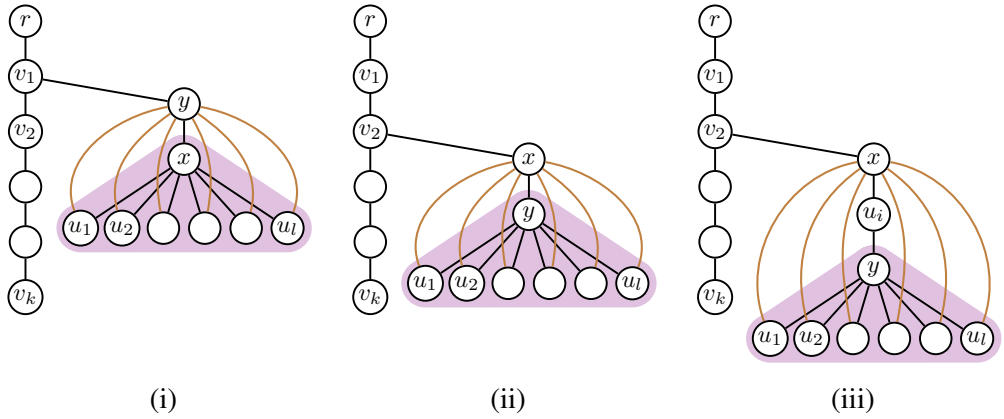

Figure 14: Worst Case Example for lower bound on maintaining DFS tree under fully dynamic edge updates.

Consider the following graph for which a DFS tree rooted at r is to be maintained under fully dynamic edge updates. There are $n/2$ vertices u_1,...,u_l that have edges to vertices x and y. The remaining $n/2-3$ vertices v_1,..., v_k are connected in form of a line as shown in Figure 14. At any point of time one of $v_1, ..., v_k$ (say v_1) is connected to either x or y. The DFS tree for the graph is shown in Figure 14 (i). Now, upon insertion of edge (v_i, x) (say $i = 2$) and deletion of edge (v_1, y) the DFS tree will transform to either Figure 14 (ii) or Figure 14 (iii). Clearly $\Omega(n)$ edges are converted from tree edge to back edge and vice-versa. This can be repeated for any v_i alternating between x and y ensuring that the new DFS tree is not exactly the same as some previous DFS tree (thus memorization of the complete tree will not help). Thus any algorithm maintaining tree edges explicitly takes $\Omega(n)$ time to handle such a pair of edge updates.

11 Conclusion

We have presented a fully dynamic algorithm for maintaining a DFS tree that takes worst case $\tilde{O}(\sqrt{mn})$ update time. This is the first fully dynamic algorithm that achieves $o(m)$ update time. In the fault tolerant setting our algorithm takes $\tilde{O}(nk)$ time to report a DFS tree, where k is the number of vertex or edge failures in the graph. We show the immediate applications of fully dynamic DFS for solving various problems such as dynamic subgraph connectivity, bi-connectivity and 2 edge connectivity. We also prove the conditional lower bound of $\Omega(n)$ on maintaining DFS tree under vertex/edge updates. Refer to 10 for details.

DFS tree has been extensively used for solving various graph problems in the static setting. Most of these problems are also solved efficiently in the dynamic environment. However, their solutions have not used dynamic DFS tree. This paper is an attempt to restore the glory of DFS trees for solving graph problems in the dynamic setting as was the case in the static setting. We believe that our dynamic algorithm

for DFS, on its own or after further improvements/modifications, would encourage other researchers to use it in solving various other dynamic graph problems.

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014.

[2] Alok Aggarwal and Richard J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[3] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.

[4] Surender Baswana and Keerti Choudhary. On dynamic DFS tree in directed graphs. In *MFCS, Proceedings, Part II*, pages 102–114, 2015.

[5] Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining DFS tree for undirected graphs. In *ICALP, Proceedings, Part I*, pages 138–149, 2014.

[6] Surender Baswana and Neelesh Khanna. Approximate shortest paths avoiding a failed vertex: Near optimal data structures for undirected unweighted graphs. *Algorithmica*, 66(1):18–50, 2013.

[7] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms*, 8(4):35, 2012.

[8] Gilad Braunschvig, Shiri Chechik, David Peleg, and Adam Sealfon. Fault tolerant additive and (μ, α)-spanners. *Theor. Comput. Sci.*, 580:94–100, 2015.

[9] Timothy M. Chan. Dynamic subgraph connectivity with geometric applications. *SIAM J. Comput.*, 36(3):681–694, 2006.

[10] Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. In *FOCS*, pages 95–104, 2008.

[11] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault tolerant spanners for general graphs. *SIAM J. Comput.*, 39(7):3403–3423, 2010.

[12] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. f-sensitivity distance oracles and routing schemes. *Algorithmica*, 63(4):861–882, 2012.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[14] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[15] Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.

[16] Ran Duan. New data structures for subgraph connectivity. In *ICALP, Proceedings, Part I*, pages 201–212, 2010.

[17] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[18] Paolo Giulio Franciosa, Giorgio Gambosi, and Umberto Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Inf. Process. Lett.*, 61(2):113–120, 1997.

[19] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[20] Daniele Frigioni and Giuseppe F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.

[21] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *29th Annual Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[22] Lee-Ad Gottlieb and Liam Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008*, pages 591–600, 2008.

[23] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30, 2015.

[24] Monika Rauch Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, 1995.

[25] Monika Rauch Henzinger. Improved data structures for fully dynamic biconnectivity. *SIAM J. Comput.*, 29(6):1761–1815, 2000.

[26] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[27] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[28] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[29] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[30] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1141, 2013.

[31] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.

[32] John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[33] John H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987.

[34] Liam Roditty. Fully dynamic geometric spanners. *Algorithmica*, 62(3-4):1073–1087, 2012.

[35] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.

[36] Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012.

[37] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), Proceedings*, pages 509–517, 2004.

[38] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[39] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[40] Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.

[41] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *STOC*, pages 112–119, 2005.

[42] Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.